Part SEVEN
Java I/O

Chapter TWENTY-THREE
Java I/O Fundamentals

Exam Objectives

Read and write data from the console.
Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io package.

I/O Streams

Java I/O (Input/Output or Reading/Writing) is a large topic, but let's start by defining what is a stream in I/O.

Although conceptually, they're kind of similar, I/O streams are not related in any way with the Stream API. Therefore, all references to streams in this chapter refer to I/O stream.

In simple words, a stream is a sequence of data.

In the context of this chapter, that sequence of data is the content of a file. Take for example this sequence of bytes:

```
...10101010001011101001010100000111...
```

When we read that sequence of bytes from a file, we are reading an *input stream*.

When we write that sequence of bytes to a file, we are writing an *output stream*.

Moreover, the content of a file can be so large that it might not fit into memory, so when working with streams, we can't focus on the entire stream at once, but only in small portions (either byte by byte or a group of bytes) at any time.

But Java not only supports streams of bytes.

In the `java.io` package, we can find classes to work with byte and character streams as well.

Files

Let's think of a *file* as a resource that stores data (either in byte or character format).

Files are organized in *directories*, which in addition to files, can contain other directories as well.

Files and directories are managed by a *file system*.

Different operating systems can use different file systems, but the files and directories are organized in a hierarchical way. For example, in a Unix-based file system, that would be:

```
/
|– home
| |– documents
| | |– file.txt
| |– music
| |– user.properties
```

Where a file or a directory can be represented by a `String` called *path*, where the value to the left of a separator character (which changes between file systems) is the parent of the value to the right of the separator, like `/home/documents/file.txt` , or `c:\home\documents\file.txt` on Windows.

In Java, the `java.io.File` class represents either a file or a directory path of a file system (there isn't a `Directory` class, at least in the standard Java I/O API, for directories):

```
File file = new File("/home/user.properties");
```

When you create an instance of the `File` class, you're not creating a new file, you are just creating an object that may represent an actual file or directory (it may not even exist yet).

But once you have a `File` object, you can use some methods to work with that file/directory. For example:

```
String path = ...
File file = new File(path);
if(file.exists()) {
    // Name of the file/directory
    String name = file.getName();
    // Path of its parent
    String parent = file.getParent();
    // Returning the time the file/directory was modified
    // in milliseconds since 00:00:00 GMT, January 1, 1970
    long millis = file.lastModified();

    // If the object represents a file
    if(file.isFile()) {
        // Returning the size of the file in bytes
        long size = file.length();
    }
    // If the object represents a directory
    else if(file.isDirectory()) {
        // Returns true only if the directory was created
        boolean dirCreated = file.mkdir();
```

```
        // Returns true only if the directory was created,
        // along with all necessary parent directories
        boolean dirsCreated = file.mkdirs();

        // Get all the files/directories in a directory
        // Just the names
        String[] fileNames = file.list();
        // As File instances
        File[] files = file.listFiles();
    }
    boolean wasRenamed = file.renameTo(new File("new file"));
    boolean wasDeleted = file.delete();
}
```

Understanding the java.io Package

Now we have reviewed the basics, we can dissect the Java I/O API.

There are a lot of classes in the `java.io` package, but in this chapter, we'll only review the ones covered by the exam.

We can start by listing four **ABSTRACT CLASSES** that are the parents of all the other classes.

The first two deal with byte streams:

- `InputStream`
- `OutputStream`

The other two deal with character streams:

- `Reader`
- `Writer`

So we can say that all the classes that have **`Stream`** in their name read or write streams of **BYTES**.

And all the classes that have **`Reader`** or **`Writer`** in their name read or write streams of **CHARACTERS**.

You shouldn't have any problem in knowing that classes that have **`Input`** or **`Reader`** in their names are for **READING** (either bytes or characters).

That all the classes that have **`Output`** and **`Writer`** in their names are for **WRITING** (either bytes or characters).

And that every class that **READS** something (or do it in a certain way), has a corresponding class that **WRITES** that something (or do it in that certain way), like `FileReader` and `FileWriter`.

However, this last rule has exceptions. The ones you should know about are **`PrintStream`** and **`PrintWriter`** (looking at the name I think it's obvious that these classes are just for output, but at least, you can tell which works with bytes and which with characters).

Next, we have classes that have **`Buffered`** in their name, which use a buffer to read or write data in groups (of bytes or characters) to do it more efficiently.

Finally, based on the idea that we can use some classes in **COMBINATION**, we can further classify the classes in the API as wrappers and non-wrappers.

Non-wrapper classes generally take an instance of `File` or a `String` to create an instance, while wrapper classes take another stream class to create an instance. The following is an example of a wrapper class:

```
ObjectInputStream ois =
    new ObjectInputStream(new FileInputStream("obj.dat"));
```

When combining classes this way, in almost all cases, it's not valid mix **OPPOSITE** concepts, like combining a `Reader` with a `Writer` or a `Reader` with an `InputStream`:

```
BufferedReader br =
    new BufferedReader(new FileInputStream("file.txt")); //error
```

This classification isn't that evident by looking at the name of the classes, but if you know what each of the classes do and think about it, you'll know if they can wrap other `java.io` classes or not.

FileInputStream

`FileInputStream` reads bytes from a file. It inherits from `InputStream`.

It can be created either with a `File` object or a `String path`:

```
FileInputStream(File file)
FileInputStream(String path)
```

Here's how you use it:

```
try (InputStream in = new FileInputStream("c:\\file.txt")) {
    int b;
    // -1 indicates the end of the file
    while((b = in.read()) != -1) {
        // Do something with the byte read
    }
} catch(IOException e) { /** ... */ }
```

There's also a `read()` method that reads bytes into an array of bytes:

```
byte[] data = new byte[1024];
int numberOfBytesRead;
while((numberOfBytesRead = in.read(data)) != -1) {
    // Do something with the array data
}
```

All the classes we'll review should be closed. Fortunately, they implement `AutoClosable` so they can be used in a `try-with-resources`.

Also, almost all methods of these classes throw `IOExceptions` or one of its subclasses (such a `FileNotFoundException`, which is pretty descriptive).

FileOutputStream

`FileOutputStream` writes bytes to a file. It inherits from `OutputStream`.

It can be created either with a `File` object or a `String` path and an optional `boolean` that indicates whether you want to overwrite or append to the file if it exists (it's overwritten by default):

```
FileOutputStream(File file)
FileOutputStream(File file, boolean append)
FileOutputStream(String path)
FileOutputStream(String path, boolean append)
```

Here's how you use it:

```
try (OutputStream out =
        new FileOutputStream("c:\\file.txt")) {
    int b;
    // Made up method to get some data
    while((b = getData()) != -1) {
        // Writes b to the file output stream
        out.write(b);
        out.flush();
    }
} catch(IOException e) { /** ... */ }
```

When you write to an `OutputStream`, the data may get cached internally in memory and written to disk at a later time. If you want to make sure that all data is written to disk without having to close the `OutputStream`, you can call the `flush()` method every once in a while.

`FileOutputStream` also contains overloaded versions of `write()` that allow you to write data contained in a byte array.

FileReader

`FileReader` reads characters from a text file. It inherits from `Reader`.

It can be created either with a `File` object or a `String` path:

```
FileReader(File file)
FileReader(String path)
```

Here's how you use it:

```
try (Reader r = new FileReader("/file.txt")) {
    int c;
    // -1 indicates the end of the file
    while((c = r.read()) != -1) {
        char character = (char)c;
        // Do something with the character
    }
} catch(IOException e) { /** ... */ }
```

There's also a `read()` method that reads characters into an array of `char`s:

```
char[] data = new char[1024];
int numberOfCharsRead = r.read(data);
while((numberOfCharsRead = r.read(data)) != -1) {
    // Do something with the array data
}
```

`FileReader` assumes that you want to decode the characters in the file using the default character encoding of the machine your program is running on.

FileWriter

`FileWriter` writes characters to a text file. It inherits from `Writer`.

It can be created either with a `File` object or a `String` path and an optional `boolean` that indicates whether you want to overwrite or append to the file if it exists (it's overwritten by default):

```
FileWriter(File file)
FileWriter(File file, boolean append)
FileWriter(String path)
FileWriter(String path, boolean append)
```

Here's how you use it:

```
try (Writer w = new FileWriter("/file.txt")) {
    w.write('-'); // writing a character
    // writing a string
    w.write("Writing to the file...");
} catch(IOException e) { /** ... */ }
```

Just like an `OutputStream`, the data may get cached internally in memory and written to disk at a later time. If you want to make sure that all data is written to disk without having to close the `FileWriter`, you can call the `flush()` method every once in a while.

`FileWriter` also contains overloaded versions of `write()` that allow you to write data contained in a `char` array, or in a `String`.

`FileWriter` assumes that you want to encode the characters in the file using the default character encoding of the machine your program is running on.

BufferedReader

`BufferedReader` reads text from a character stream. Rather than read one character at a time, `BufferedReader` reads a large block at a time into a buffer. It inherits from `Reader`.

This is a wrapper class that is created by passing a `Reader` to its constructor, and optionally, the size of the buffer:

```
BufferedReader(Reader in)
BufferedReader(Reader in, int size)
```

`BufferedReader` has one extra read method (in addition to the ones inherited by `Reader`), `readLine()`. Here's how you use it:

```
try ( BufferedReader br =
     new BufferedReader( new FileReader("/file.txt") ) ) {
   String line;
   // null indicates the end of the file
   while((line = br.readLine()) != null) {
      // Do something with the line
   }
} catch(IOException e) { /** ... */ }
```

When the `BufferedReader` is closed, it will also close the `Reader` instance it reads from.

BufferedWriter

`BufferedWriter` writes text to a character stream, buffering characters for efficiency. It inherits from `Writer`.

This is a wrapper class that is created by passing a `Writer` to its constructor, and optionally, the size of the buffer:

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int size)
```

`BufferedWriter` has one extra write method (in addition to the ones inherited by `Writer`), `newLine()`. Here's how you use it:

```
try ( BufferedWriter bw =
     new BufferedWriter( new FileWriter("/file.txt") ) ) {
   w.newLine("Writing to the file...");
} catch(IOException e) { /** ... */ }
```

Since data is written to a buffer first, you can call the `flush()` method to make sure that the text written until that moment is indeed written to the disk.

When the `BufferedWriter` is closed, it will also close the `Writer` instance it writes to.

ObjectInputStream/ ObjectOutputStream

The process of converting an object to a data format that can be stored (in a file for example) is called *serialization* and converting that stored data format into an object is called *deserialization*.

If you want to serialize an object, its class must implement the `java.io.Serializable` interface, which has no methods to implement, it only tags the objects of that class as serializable.

If you try to serialize a class that doesn't implement that interface, a `java.io.NotSerializableException` (a subclass of `IOException`) will be thrown at runtime.

`ObjectOutputStream` allows you to serialize objects to an `OutputStream` while `ObjectInputStream` allows you to deserialize objects from an `InputStream`. So both are considered wrapper classes.

Here's the constructor of the `ObjectOutputStream` class:

```
ObjectOutputStream(OutputStream out)
```

This class has methods to write many primitive types, like:

```
void writeInt(int val)
void writeBoolean(boolean val)
```

But the most useful is `writeObject(Object)`. Here's an example:

```
class Box implements java.io.Serializable {
   /** ... */
}
...
try( ObjectOutputStream oos =
       new ObjectOutputStream(
         new FileOutputStream("obj.dat") ) ) {
    Box box = new Box();
```

```
    oos.writeObject(box);
} catch(IOException e) { /** ... */ }
```

To deserialize the file `obj.dat`, we use `ObjectInputStream` class. Here's its constructor:

```
ObjectInputStream(InputStream in)
```

This class has methods to read many data types, among them:

```
Object readObject()
      throws IOException, ClassNotFoundException
```

Notice that it returns an `Object` type. Thus, we have to cast the object explicitly. This can lead to a `ClassCastException` thrown at runtime. Note that this method also throws a `ClassNotFoundException` (a checked exception), in case the class of a serialized object cannot be found.

Here's an example:

```
try (ObjectInputStream ois =
        new ObjectInputStream(
          new FileInputStream("obj.dat") ) ) {
    Box box = null;
    Object obj = ois.readObject();
    if(obj instanceof Box) {
        box = (Box)obj;
    }
} catch(IOException ioe) { /** ... */ }
catch(ClassNotFoundException cnfe) {
   /** ... */
}
```

Two important notes. When deserializing an object, the constructor, and any initialization block are not executed, Second, `null` objects are not serialized/deserialized.

PrintWriter

`PrintWriter` is a subclass of `Writer` that writes formatted data to another (wrapped) stream, even an `OutputStream`. Just look at its constructors:

```
PrintWriter(File file)
    throws FileNotFoundException
PrintWriter(File file, String charset)
    throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(String fileName) throws FileNotFoundException
PrintWriter(String fileName, String charset)
    throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
```

By default, it uses the default charset of the machine you're running the program, but at least, this class accepts the following charsets (there are other optional charsets):

- US-ASCII
- ISO-8859-1
- UTF-8

- `UTF-16BE`
- `UTF-16LE`
- `UTF-16`

As any `Writer`, this class has the `write()` method we've seen in other `Writer` subclasses, but it overwrites them to avoid throwing an `IOException`.

It also adds the methods `format()`, `print()`, `printf()`, `println()`.

Here's how you use this class:

```
// Opens or creates the file without automatic line flushing
// and converting characters by using the default character encoding
try(PrintWriter pw = new PrintWriter("/file.txt")) {
    pw.write("Hi"); // Writing a String
    pw.write(100); // Writing a character

    // write the string representation of the argument
    // it has versions for all primitives, char[], String, and Object
    pw.print(true);
    pw.print(10);

    // same as print() but it also writes a line break as defined by
    // System.getProperty("line.separator") after the value
    pw.println(); // Just writes a new line
    pw.println("A new line...");

    // format() and printf() are the same methods
    // They write a formatted string using a format string,
    // its arguments and an optional Locale
    pw.format("%s %d", "Formatted string ", 1);
    pw.printf("%s %d", "Formatted string ", 2);
    pw.format(Locale.GERMAN, "%.2f", 3.1416);
    pw.printf(Locale.GERMAN, "%.3f", 3.1416);
} catch(FileNotFoundException e) {
    // if the file cannot be opened or created
}
```

You can learn more about format strings for `format()` and `printf()` in https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html

Standard streams

Java initializes and provides three stream objects as `public static` fields of the `java.lang.System` class:

- `InputStream System.in`
  The standard input stream (typically the input from the keyboard)
- `PrintStream System.out`
  The standard output stream (typically the default display output)
- `PrintStream System.err`
  The standard error output stream (typically the default display output)

`PrintStream` does exactly the same and has the same features that `PrintWriter`, it just works with `OutputStreams` only.

The following example shows how to read a single character (a byte) from the command line:

```
System.out.print("Enter a character: ");
try {
    int c = System.in.read();
```

```
} catch(IOException e) {
    System.err.println("Error: " + e);
}
```

Or to read `String` s:

```
BufferedReader br =
    new BufferedReader(new InputStreamReader(System.in));
String line = br.readLine();
// Or using the java.util.Scanner class
Scanner scanner = new Scanner(System.in);
String line = scanner.nextLine();
```

java.io.Console

Since Java 6, we have the `java.io.Console` class to access the console of the machine your program is running on.

You can get a reference to this class (is a singleton) with `System.console()` .

But keep in mind that if you program is running in an environment that doesn't have access to a console (like an IDE or if your program is running as a background process), `System.console()` will return null.

With the `Console` object, you can easily read user input with the `readLine()` method and even read passwords with `readPassword()` .

For output, this class has the `format()` and `printf()` methods that work just like the ones of `PrintWriter` .

Finally, the methods `reader()` and `writer()` return an instance of `Reader` and `Writer` respectively:

```
Console console = System.console();
// Check if the console is available
if(console != null) {
    console.writer().println("Enter your user and password");
    String user = console.readLine("Enter user: ");
    // readPassword() hides what the user is typing
    char[] pass = console.readPassword("Password: ");
    // Clear password from memory by overwriting it
    Arrays.fill(pass, 'x');
}
```

`readPassword()` returns a `char` array so it can be totally and immediately removed from memory (`String` s live in a pool in memory and are garbage collected, so an array is safer).

Key Points

- An I/O Stream is a sequence of data that represents the content of a file.
- An input stream is for reading and an output stream is for writing.
- In the `java.io` package, we can find classes to work with byte and character streams.
- There are four main `abstract` classes from which the rest of the classes extend from `InputStream` , `OutputStream` , `Reader` , `Writer` .
- `java.io` classes can be classified as:
    - Either for byte streams or character streams

    - Either for input or for output
    - Either wrappers or non-wrappers
- Java initializes and provides three stream objects as `public static` fields of the `java.lang.System` class:
    - `InputStream System.in`
      The standard input stream (typically the input from the keyboard)
    - `PrintStream System.out`
      The standard output stream (typically the default display output)
    - `PrintStream System.err`
      The standard error output stream (typically the default display output)
- The following table summarizes the classes reviewed in this chapter:

Class	Extends From	Main Constructor Arguments	Main Methods	Description
`File`	–	• `String`	• `exists` • `getParent` • `isDirectory` • `isFile` • `listFiles` • `mkdirs` • `delete` • `renameTo`	Represents a file or directory.
`FileInputStream`	`InputStream`	• `File` • `String`	• `read`	Reads file content as bytes.
`FileOutputStream`	`OutputStream`	• `File` • `File, boolean` • `String` • `String, boolean`	• `write`	Writes file content as bytes.
`FileReader`	`Reader`	• `File` • `String`	• `read`	Read file content as character.
`FileWriter`	`Writer`	• `File` • `File, boolean` • `String` • `String, boolean`	• `write`	Write file content as character.
`BufferedReader`	`Reader`	• `Reader` • `Reader, int`	• `readLine` • `read`	Reads text to a character stream, buffering characters for efficiency.

`BufferedWriter`	`Writer`	• `Writer` • `Writer, int`	• `newLine` • `write`	Writes text to a character stream, buffering characters for efficiency.
`ObjectInputStream`	`InputStream`	• `InputStream`	• `readObject`	Deserializes primitive data and objects.
`ObjectOutputStream`	`OutputStream`	• `OutputStream`	• `writeObject`	Serializes primitive data and objects
`PrintWriter`	`Writer`	• `File` • `OutputStream` • `String` • `Writer`	• `format` • `print` • `printf` • `println`	Writes formatted data to a character stream.
`Console`	–	–	• `readLine` • `readPassword` • `format` • `printf`	Provides access to the console, if any.

Self Test

1. Which of the following is a valid way to create a `PrintWriter` object?
A. `new PrintWriter(new Writer("file.txt"));`
B. `new PrintWriter();`
C. `new PrintWriter(new FileReader("file.txt"));`
D. `new PrintWriter(new OutputStream("file.txt"));`

2. Given:

```
try (Writer w = new FileWriter("/file.txt")) {
    w.write('1');
} catch(IOException e) { /** ... */ }
```

Which of the following is the result of executing the above lines if the file already exists?
A. It overwrites the file
B. It appends `1` to the file
C. Nothing happens since the file already exists
D. An `IOException` is thrown

3. Which of the following is the type of the `System.in` object?
A. `Reader`
B. `InputStream`
C. `BufferedReader`
D. `BufferedInputStream`

4. Given:

```
class Test {
    int val = 54;
}
public class Question_23_4 {
    public static void main(String[] args) {
        Test t = new Test();
        try (ObjectOutputStream oos =
  new ObjectOutputStream(new FileOutputStream("d.dat"))) {
            oos.writeObject(t);
        } catch (IOException e) {
            System.out.println("Error");
        }
    }
}
```

Which of the following is the result of executing the above lines?
A. Nothing is printed, the class is serialized in `d.dat`
B. Nothing is printed, but the class is not serialized
C. `Error`
D. An runtime exception is thrown

5. What does the `flush()` method do?
A. It marks the stream as ready to be written.
B. It closes the stream.
C. It writes the data stored in disk to a cache.
D. It writes the data stored in a cache to disk.

Open answers page

---

22. Time Zones and Daylight Savings 24. NIO.2